ASIC Data Collection Using FPGA Interface

Tutorial on how to quickly collect data from ASIC using FPGA

Shane Kelly and Mohammad Tehranipoor
shane.p.kelly@uconn.edu , tehrani@engr.uconn.edu

This article will detail an experiment setup to quickly collect data from application specific integrated circuit (ASIC) test chips. The topics covered are Serial UART communication for field programmable gate array (FPGA), Linux serial communication, VHDCI and VmodBB wiring and FPGA Programming. FPGA programming includes Controller Design, FPGA clock manipulation, UART communication module and VHDC usage.

This article assumes the reader has basic knowledge of VHDL and some experience with either Bash or Python.

Tutorial Outline

1. Introduction
2. FPGA Controller Design
   - VHDCI and the VmodBB
   - Clock Manipulation
   - UART Communication Component
   - State Machine Design
3. Serial Communication
   - Bash Serial IO
   - Python Serial IO
4. Trojan Detection Example

Prerequisites and Requirements

- Fabricated ASIC
- Vmod Breadboard.
- FPGA with VHDCI and UART-USB interface.
- Required Software
  - Linux based operating system.
  - Xilinx ISE web pack(Free)[3]
  - Digilent Adept(Free)[4]
- Knowledge of VHDL
- Knowledge of FPGA programing(FPGA Programing Tutorials from Digilent)[5]

1 Introduction

Implementation of theoretical ideas can be a difficult and daunting task because of the resources needed for fabrication and the potential time data collection takes. To address the concern of fabrication, MOSIS Educational Program(MEP) offers free fabrication for both instructional and research purposes[1]. This article addresses the concern of data collection time and procedure for digital devices by suggesting a fast data collection technique. The readers are referred to MOSIS webpage to learn more about the available technologies, fabrication dates, design flow, etc.

The essential idea behind this collection method is to put sensors on the test chip to measure important parameters instead of using expensive or time consuming equipment like an oscilloscope. With a chip designed with sensors, one can quickly collect data by simply running the tests, reading the data from the senor and reporting the data. With the FPGA and VmodBB set up shown below, multiple tests can be conducted on one chip in less than a few second. An example of using on-chip sensors to measure circuit parameters is shown in figure 1[2]

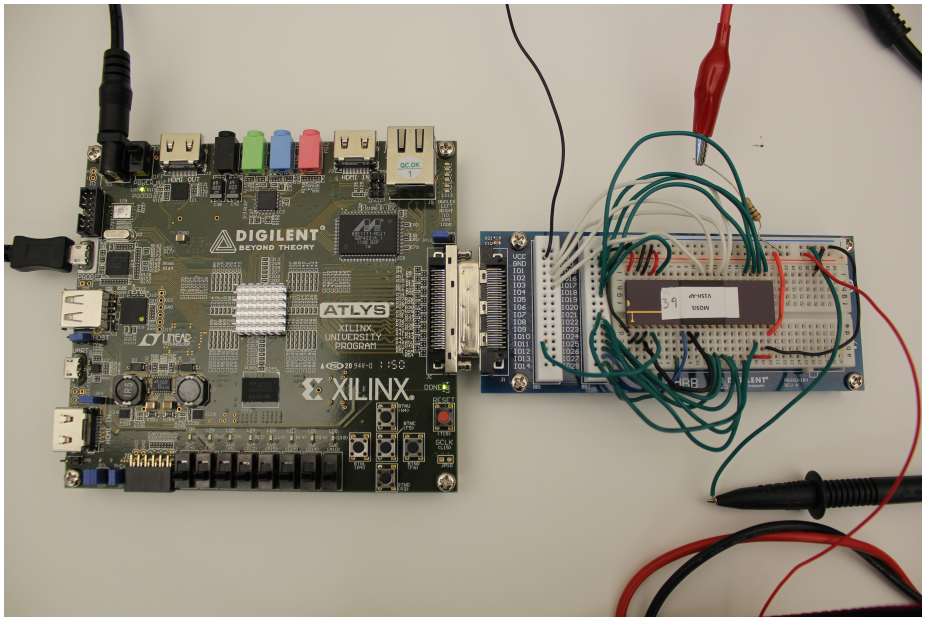

Figure 1: Test Setup

This article describes in detail the set up shown above, so future students can follow this tutorial and easily set up a similar experiment. In short the ASIC is wired to a bread board, which is connected to an FPGA using a very high density cable interconnect(VHDCI). A controller on the FPGA then sends control signals and reads data from the ASIC. This data is then sent to a computer via, USB UART. This article describes VmodBB wiring, the FPGA clock manipulation, FPGA UART design and serial communication. This article uses the Spartan-6 FPGA on an ATLYS board as an example but one can use any FPGA or board as long as it has a UART serial interface and VHDCI.

2 FPGA Controller Design

The first step in creating an quick test environment is setting up the FPGA to send and receive information with both the computer and the ASIC. Begin by creating a new project in the XILINX ISE and create a new `".vhd"` file and name it toplevl. This will be the main file to be synthesised to be programmed to the FPGA. Its inputs should be the output signals of the ASCI and the FPGA clock and the output signals should be the UART TX signal(1 bit) and the input signals to the ASIC. This module will include a state machine to control the signals being sent and received from the FPGA. It also contains components to control the clock, to control the UART signal and receive button inputs. The first section, subsection 2.1, will begin describing the inputs and outputs of the top file and describes the VHDCI interface. subsection 2.2 describes how to change the clock frequency and phase being sent to the ASIC, subsection 2.3 describes how to use the UART component to send signals to the computer from the FPGA and subsection 2.4 describes the state machine used to control the inputs and outputs to the ASIC and UART interface.

For tutorials on using ISE web-pack and programming FPGAs see the Digilent website[5]. For Linux the ISE web-pack is identical but programming the FPGA can be a little different. From terminal one can list the FPGAs connected to their computer use

```
djtgcfg enum
```

And to program a FPGA with `toplvl.bit` use

```
sudo djtgcfg prog -i 0 -d Atlys -f toplvl.bit
```

Where `-i 0` specifies that you want to use the first index in JTAG scan chain to program, `-d Atlys` specifies to program the Atlys device connected to your computer and `-f toplvl.bit` specifies that you want to program the file topvlv.bit to the FPGA.

2.1 VHDCI and the VmodBB

To control the ASIC signals from the FPGA, begin by placing the ASIC on the VmodBB and wiring the ground(highlighted in black in Figure 2 and power slots(highlighted in red in Figure 2 to the power and ground pins of the ASIC . The blue cap on the ATLYS FPGA selects whether 3.3V or 2.5V is supplied to VDD. You can also use an external power supply as shown in Figure 2.

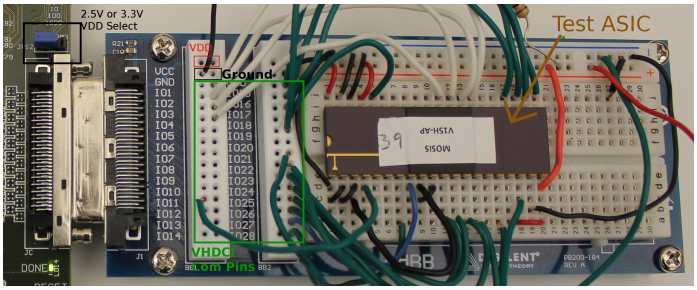

Figure 2: VmoddBB Connections. Green Highlights the IO slots for the VHDCI. The slots highlighted in black are the ground slots and the slots highlighted in red are the power supply slots.

Next wire the inputs and outputs of the ASIC to the bread board slots labeled IO1-IO28. Wire the constant inputs to either VDD or ground to save the VHDCI IOs for other signals.

Setting Up Your UCF File The next step is to assign the ASIC input and output signals in `toplevel.bit` to the physical VHDCI ports on the FPGA. This is done using a user constraint file(UCF), which can be found on the bottom of the web page for your FPGA on the Digilent website. If you have an ATLYS board you can use the general UCF I supplied with this tutorial which has comments explaining which location corresponds to which `IO#` slot on the VmodBB. If you aren't using the ATLYS board you can determine the UCF nets that the VHDCI uses by looking at the VmodBB schematic[6]. The schematic will relate the `IO#` name to either a `IO#-N` or `IO#-P` name which will be labeled in the UCF file. An example that routes the toplevel signal, "TLsignal", to the slot labeled "IO15" on the VmodBB is shown here:

```
NET "TLsignal" LOC ="V16"; # VmodBB Hole Name:IO15 Bank = 2, Pin IO_L2N_CMPMOSI,Sch name = EXP-IO1_N
```

This is an example for the ATLYS board using the UCF file I supplied. For other UCF files the comment `"VmodBB Hole Name:IO#"` will not be there and you will need to compare the `"Sch name"` to the VmodBB name shown in the VmodBB schematic[6]. Make sure to comment out any unused signals.

At this point you should assign the buttons and switches on the FPGA to the VHDCI signals so you can manually test whether your set up works. To do this you need to assign the button and switches locations to the input signal names in the UCF.

```
NET "TL_btn_signal" LOC = "T15"; # Bank = 2, Pin name = IO_L1N_M0_CMPMISO_2, Sch name = M0/RESET
```

Then in the top level file connect the input button signals to the output VHDCI signals.

```
VHDCI_OUT<=TL_btn_signal;
```

Then in the UCF assign the output VHDCI signals of in the top level file to VHDCI loc on FPGA.

```
 NET "VHDCI_OUT" LOC = "V8";  # VmodBB Hole Name:IO28 Bank = 2,  Pin name = IO_L41N_VREF,Sch name = EXP-IO14_N
```

Now button 0, which happens to be the reset button on the Atlys board, controls the VmodBB pin 28. You should be able to do this with what ever signals you like. The only one that will give you difficulty is the clock signal. The next section goes over how to generate the clock signal you would like to send to the FPGA.

2.2 Clock Manipulation

For manual testing, it was suggested that you connect these signals to the buttons or switches of the FPGA and manually send the control signals. The one signal you likely can not do this is with the clock signal. This section will describe how to create the desired clock signal and how to buffer it to the ASIC.

Depending on the type of clock signal you want and the components available to you on your FPGA you will need to use either the XILINX Digital Clock manager(DCM) or the XILINX Digital Phased-Locked Loop(DPLL). In general if you don't care about the phase of the clock you can use the Digital Clock Manager to adjust the signal to the correct clock frequency, otherwise you will need to use the PLL. More more information on both of these see there documentation: DCM or DPLL[7][8]. Since using these tools can be complex I suggest using the XILINX clock wiz core geneartor.

Xilinx Clocking Wizard You can access the core generator by clicking on Tools»Core Generator in the XILINX Web Pack ISE. This opens up he Xilinx CORE generator manager.

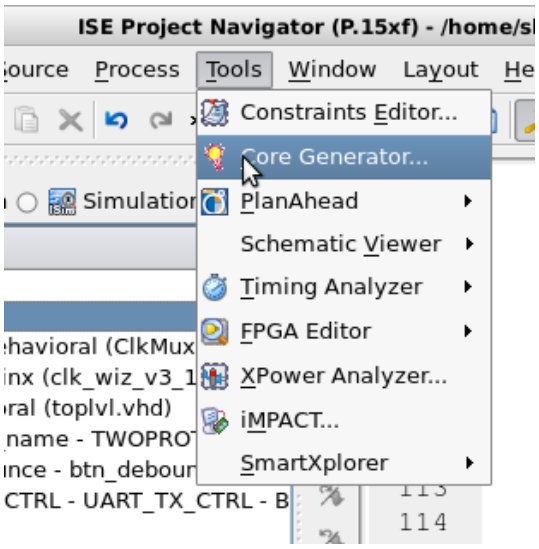

Figure 3: Opening Core Generator

From here, navigate the IP Catalog to FPGA Features and Design»Clocking>Clocking Wizard under the View by Function tab.

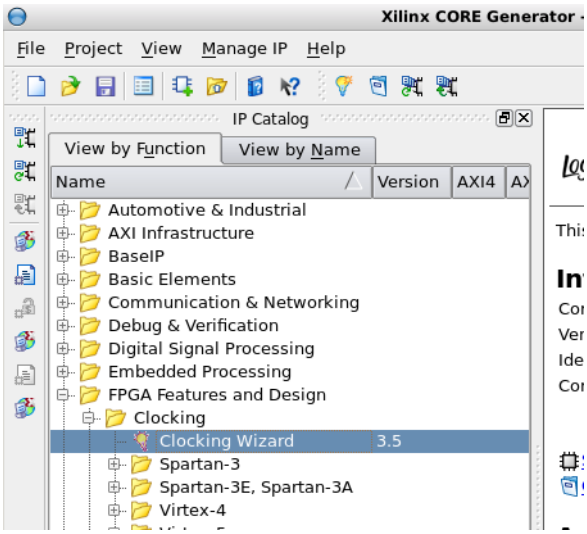

Figure 4: Opening Clocking Wizard

This will open up a Clocking Wizard Window which has 6 pages to customize your clock. On the first page you can name the module and describe the Clock input signal. If you are using the spartan 6 Atlys board the input frequency should be 100 and the Input Jitter should be 0.01.

On the second page specify the Type of clock you want by changing the Output Frequency,Phase and Duty Cycle.

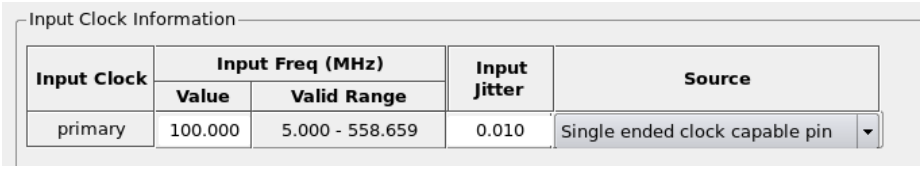

Figure 5: Page 1 Settings

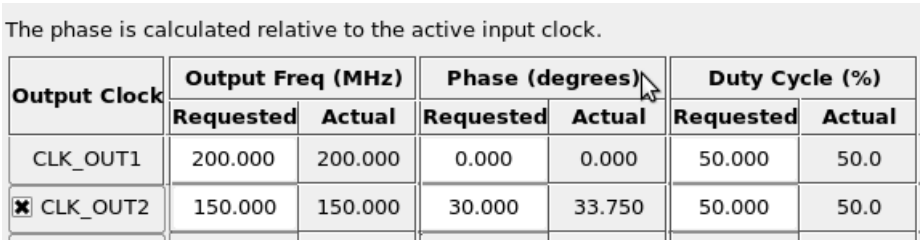

Figure 6: Page 2 Settings

The third page describes control signals such as sync and reset and can be unchecked. The other pages are a summery of what will be generated. After Generation you can click "View HDL Instantiation Template" to view a template for the VHDL code that you will need to instantiate this component in the top level file. After you have instantiated the clock-wiz component, connect the top level clock input signal to the FPGA clock using the UCF file. For the Atlys board use

```
# clock pin for Atlys rev C board
NET "TL_WIZ_INPUT"  LOC="L15"; # Bank =1, Pin name =IO_L42P_GCLK7_M1UDM, Type = GCLK, Sch name =GCLK
```

These lines should already be in the master UCF and all you have to change is the clk signal name to what ever name you used for the clock signal input in your top level file. Then connect the ASIC clock signal to the correct net as you did for a normal signal in the previous section.

```
NET "ASIC_clk" LOC ="V15"; # VmodBB Hole Name:IO16 Bank = 2, Pin *IO_L5N,       Sch name = EXP-IO2_N
```

Where IO16 is the slot of the VmodBB that is connected to ASIC clock and `ASIC_clk` is the name of the output signal for the `toplevel.vhd` file. At this point you should be able to manually test the ASIC using the buttons and switches assigned in the previous section.

Note on Clock Buffering If you are having problems with the clock signal or errors about clock forwarding there may be a problem with buffering the clock to th IO. To fix this you can add an ODDR2 buffer before you output the signal. You can do this with the following code:

```
Library UNISIM;
use UNISIM.vcomponents.all;
ODDR2_50 : ODDR2
generic map(
        DDR_ALIGNMENT => "NONE", -- Sets output alignment to "NONE", "C0", "C1"
        INIT => '0', -- Sets initial state of the Q output to '0' or '1'
        SRTYPE => "SYNC") -- Specifies "SYNC" or "ASYNC" set/reset
port map (
```

```
Q => clkIO50, -- 1-bit output data
C0 => clk_50, -- 1-bit clock input
C1 => not clk_50, -- 1-bit clock input
CE => '1', -- 1-bit clock enable input
D0 => '1', -- 1-bit data input (associated with C0)
D1 => '0', -- 1-bit data input (associated with C1)
R => '0', -- 1-bit reset input
S => '0' -- 1-bit set input
);
```

where `clkIO50` is the output of the buffer and `clk_50` is the input.

2.3 UART Communication Component

If you have been following the tutorial up until now you have a top file with inputs and outputs connected to the FPGA, You have possibly wired some of the inputs to switches and buttons on the FPGA and you have set up the clock to be sent to the ASIC. The next step is to set up the UART control module to communicate with the computer. With most FPGAs that have UART interfaces, Digilent provides an example that uses UART in the support documents on the Digilent page[9]. The example for the ATLYS board can be found here[10]

From the Digilent example, add the `UART_TX_CTRL.vhd` file to your project by clicking project»Add Copy of Source and then navigate to where you saved that file. First, instantiate the module in the top level file and then connect the $UART_TX$ to the correct location as they do in *pin.ucf* in the example. For the Atlys board this would be done with

```
NET "TL_UartTx_Signal" LOC = "B16"; # Bank = 0, Pin name = IO_L66P_SCP1, Sch name = USBB-TXD
```

This is the signal that the computer will receive and process. For the `UART_TX_CTRL` component, the ready signal is raised high when the component is ready to send information. The send signal tells the `UART_TX_CTRL` component to send the data present at the data signal and should not be raised high unless ready is high. Finally, connect a 100MHz clock(Spartan 6 FPGA clock) to the clock signal. Since the `UART_TX_CTRL` component can only send 8 bits of data at time, its up to you to configure how to send the data collected from the ASIC. An important question to consider when preparing the output data, is how the computer will read that data? If the output of the ASIC is a number, you may want to consider sending that number in ASCI hex, one digit at a time. This way the file you save the data to can be viewed in a text document.

For example, assume the output of each trial is a 16 bit number. In this situation you could send the 16 bit number across as a 4 digit hex number. To do this you will first need to convert each 4 bit digit to its ASCI hex value. A simple way of doing this is with a "WHEN" statement in VHDL.

```
data_ic<=EXP_IO_P(3 downto 0);
data_touart <=
        x"0A" when Enter='1' else
        x"2C" when debounced_btn(3)='1' or Comma='1' else
        x"46" when data_ic="1111" else
        x"45" when data_ic="1110" else
        x"44" when data_ic="1101" else
        x"43" when data_ic="1100" else
        x"42" when data_ic="1011" else
        x"41" when data_ic="1010" else
```

```
x"39" when data_ic="1001" else
x"38" when data_ic="1000" else
x"37" when data_ic="0111" else
x"36" when data_ic="0110" else
x"35" when data_ic="0101" else
x"34" when data_ic="0100" else
x"33" when data_ic="0011" else
x"32" when data_ic="0010" else
x"31" when data_ic="0001" else
x"30" when data_ic="0000";
```

Where `Data_IC` is the 4 bit signal to be sent and `data_touart` is the signal connected to the UART component. The code above will also send a new line symbol when the enter signal is enabled and will send a comma when the comma signal is enabled. This way the state machine that controls the IO signals connected to the ASIC will also be able to create a comma delimited file which can easily be parsed by most programing languages.

You should always use the button debounced component that came with the Digilent example when testing the UART component with buttons. This component requires a button to be released or held down for a certain number of clock cycles to register an action. This will eliminate spurious input which may cause problems with the UART send signal.

2.4 State Machine Design

At this point in the tutorial your set up is ready to send input to the ASIC via VHDCI, receive the output of the ASIC via VHDCI and then send that data to the computer via UART. The finial step is to automate the process with a state machine. Before doing this you should connect the important signals to the switches and buttons on the FPGA and manually test your setup.

Assuming now that everything is working you are now ready to design a state machine to the top level file. The state machine should manage the signals being sent to the ASIC, read the output of the ASIC at the end of each trial and prepare the data to be sent to the computer using the UART component. A general state diagram is shown in figure 7.

First the FPGA starts in the ready state waiting for the user to press a button that enables the start signal. When the start signal is enabled the FPGA moves to the reset state, applies the reset signal to the ASIC and moves onto the next State. The FPGA waits for a number of clock cycles R for the chip to finish reseting and then moves on to the start state. In the start state the FPGA applies the start signal to the ASIC and moves on to the wait state to wait to the experiment to complete. After the S clock cycles it takes to complete the experiment the FPGA moves onto the Read Data data state where data that is to be sent to the computer. In this state the FPGA waits for the UART component to signal ready to send data. When it receives the Ready signal, the state machine applies the send signal to 1 and moves to the next send data state. As soon as the state machine enters the send data state it sets send to 0 so it was only active for one clock cycle. The FPGA then waits until the UART component was done sending or Ready is set to '1' again. Finally The state machine either prepares new inputs and moves back to Reset or goes back to Ready and finishes.

Of course this is a general description and the states selecting new inputs and sending the Data via UART will likely be more complex and composed of states them selves. For example if you are trying to send the 16 bit number

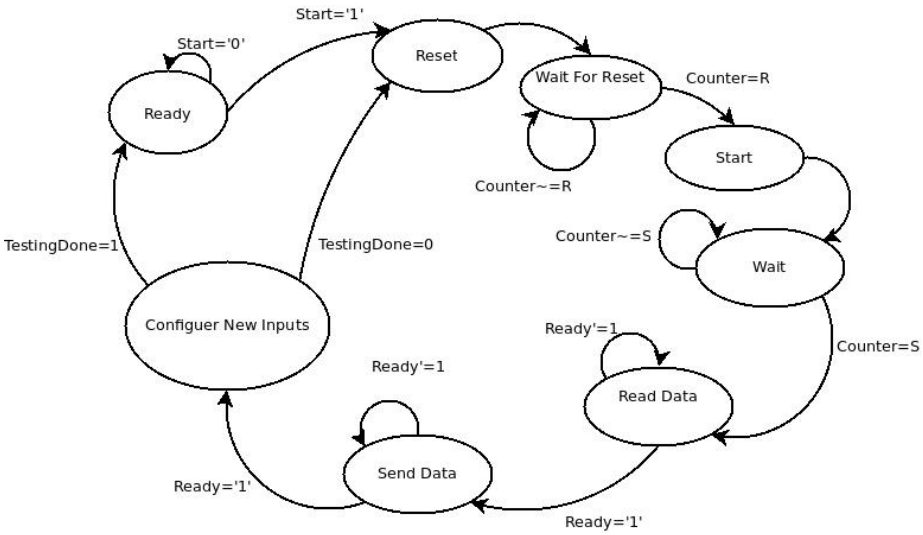

Figure 7: FPGA Controller State Machine

described in the previous section you will need 8 states for sending the data instead of only 2. Two states are needed for sending data 1 byte of data. The Read Data state prepares the data at the input to the UART component and waits for the UART component to signal Ready. The Send Data state waits for the data to finish sending before preparing new data.

3 Serial Communication

The section goes over receiving serial data through a USB port. All though this is possible to do in windows it tends to be more difficult locating the serial port to be used. For that reason this part of the tutorial will be Linux specific but if you can find the serial port the python tutorial will be the same for both Linux and Windows. The first thing in reading data form a serial connection is to determine the driver responsible for managing the connection. This can be done by navigating to `\dev` and look for a new file to appear when plugging in the serial USB. The file will likely contain 'tty' and will disappear if you remove the USB cable connected to the UART port on the FPGA. The next few sections will address how to use this driver to obtain data from the USB-UART connection. The method that uses bash tends to be quicker but is less flexible then the python method.

3.1 Bash Serial IO

The first thing to do is to set the baud rate so the computer can properly interpret the signal it receives from the FPGA. To begin, check what the current baud rate is:

```
stty --file=/dev/ttyACM0 -a
```

Where ttyACM0 is an example name of the driver you found before. This command will return information about the file you specified and on the first line will be some thing like: "speed 9600 baud". If the number is 9600 then you have the correct baud rate set to communicate with the UART TX component of the fpga. If the number is anything else, then you can set it to the correct baud rate by typing the command

```
screen /dev/ttyACM0 9600
```

Now type control-C followed by

```
:quit
```

To quit screen. The finial thing to do is to view the information being sent to the serial port. This can be done simply with the cat command

```
cat /dev/ttyACM0
```

to display what is being sent to serial to terminal. This command will work in real time so as you send more data from the FPGA this data will appear in the terminal. To save the data to a file

```
cat /dev/ttyACM0>file_name.csv
```

When using the command this way, the data will not be displayed in the terminal as you send more data. Since the FPGA can't fully close the driver, both of these commands can must be stopped by terminating the process or typing control-C. For more information about using the command line to collect data from a serial port look here[11].

3.2 Python Serial IO

Another way to read data from the serial port is to use the python module pySerial. First thing you need to do is import this module with

```
import serial
```

Then open the serial port with

```
port = serial.Serial('/dev/ttyACM0',9600,timeout=1)
```

Where port is now an object you can use to read from the serial connection, `/dev/ttyACM0'` is an example name for a driver you would have found before and 9600 is the baud rate. Now you can read from port like you would a file you normally opened with commands like `'port.readline()'` (read a line at a time) or `'port.read()'`(read all the data that has been sent). It is important to note that you will only be able to read data sent to the port after this method has been executed. So your program will need to open the port then ask if the data has been sent and then read from the port. An example that writes the data received to a file called TestData is shown below.

```
import serial
port=serial.Serial('/dev/ttyACM0',9600,timeout=1)
raw_input("Press Enter After Collecting...")
total=''
current =port.readline()
while(current!=''):
        total=total+current
        current=port.readline()
port.close()
f=open('TestData','w')
f.write(total)
f.close()
```

Doing this in python can have many advantages. The first being that you don't have to type control-c to stop data collection breaking up control flow. The second things is now that you wrote the data to a file, you can now do operations on it, either in python or call other programs from python.

4 Trojan Detection Example

This test set up was used in testing a Ring Oscillator Network's ability to detect the addition of malicious circuitry. Briefly summarized, the frequency of a ring oscillator is dependent on the power supplied to it. If additional circuitry is added near a ring oscillator, it will draw power away from the ring oscillator dropping the ring oscillator frequency and allow for trojan detection. To test this a 90nm ASIC with 8 ring oscillators and 31 Trojans was fabricated using the MEP research program. The frequency of each RO was counted by a counter and stored in a 16 bit register. A multiplexer was used to select 4 bit sections of the 16 bit register to be routed to the 4 output pins. Another multiplexer was used to select which ring oscillators to measure and another one was for which Trojan, if any, to activate.

VmodBB wiring.The first thing done wiring the 10 inputs to the 3 multiplexers, the reset pin, the start pin and the clock signal were on the ASIC to 13 IO slots on the VmodBB. The inputs that stayed constant during data collection, like the test mode inputs, were wired to either ground or VDD. Then the 4 bit output was wired to another 4 IO slots and all other outputs were left unconnected. As described in subsection 2.1, the top level file was set up with 2 switches wired to the output multiplexer, 3 switches where wired to ring oscillator multiplexer, the lats three switches where wired to the first 3 inputs to the Trojans and the last inputs to the Trojans where set to '0' just for testing purposes. The outputs were wired to 4 led's and the start and input signal were wired to buttons.

Clock Manipulation and UART Communication.Using the clocking wizard we set up a 200MHz clock signal and buffered it to the input using ODR2 as described in subsection 2.2. We then set up the UART module to send the 4bit output of the ASIC as a hex number using the when statement described in subsection 2.3. We then connected one of the buttons to the enter signal, another button to the comma signal and the last button to the send signal. At this point we use the `cat`,as described in subsection 3.1, to test if we could collect data from each oscillator and each trojan and send it to computer manually.

State Machine Design.The finial step for FPGA design was to automate the process. The specific state machine used in shown in Figure 8 and the top level file with the state machine implemented using a process statement is supplied with this document as `toplevel.vhd`. The signal RO is the 3 bit signal being sent to the ring oscillator input and the signal TR is the 5 bit signal being sent to the trojan multiplexer. The state machine starts in the same way the general one shown in Figure 7 does. It starts in the ready state, prepared to collect data from ring oscillator 1 with all Trojans disabled and waits until the Go signal(wired to a button) is enabled. After Go is enabled the state machine resets the chip and starts the chip. It sends the data in a similar way described in subsection 2.4 but now there are two states to send each 4bit section of the 16bit register in the ASIC. So by the time the UART READY signal is enabled after state `Send0`, the computer has received the number of cycles the ring oscillator made in a 4 digit ASCI hex number. At this point the state machine decides to send either an end of line character or comma.

After It finishes sending either one it moves to the prepare data state where

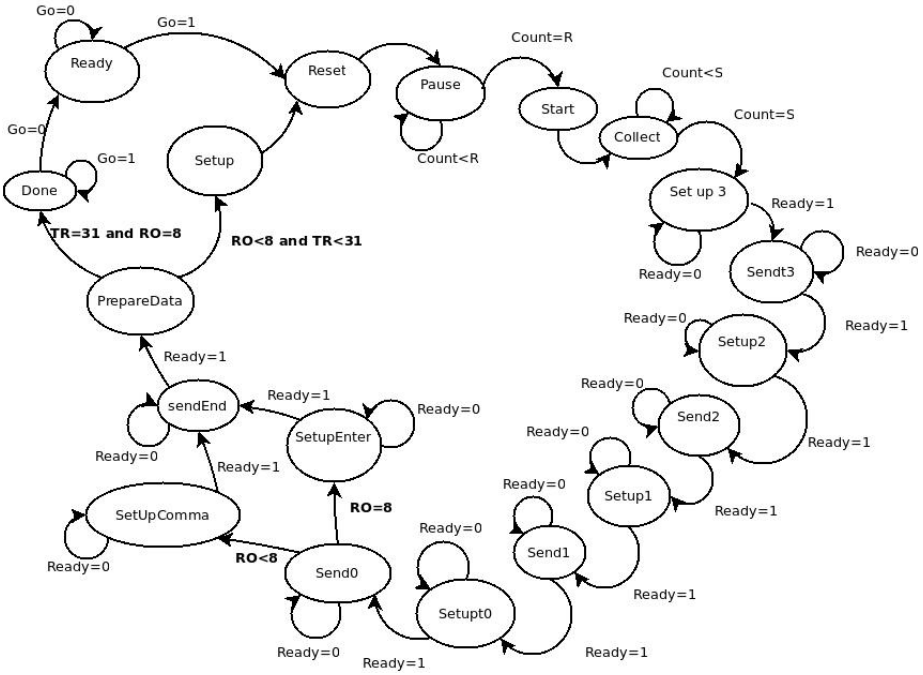

Figure 8: FSM for Trojan Detection Example

it decides what trojan to enable and what ring oscillator to collect data from. It does this by incrementing the RO signal until all ring oscillators have been measured for a single Trojan Multiplexer input. Once the 8th ring oscillator has been collected, RO is reset to 0 and TR is incremented. This continues until TR=31 and RO=8 when all the data from the chip has been collected and the process is complete. At this point the state machine enters the done state and waits for the Go signal to return to 0. This way the state machine doesn't repeat these trials again because the "Go button" was held down to long. By having a comma sent after each collection where RO<8 and sending an end of line character when RO=8, the data sent to computer will appear as a comma delimited file where the i^{th} row represents $TR = i$ and the j^{th} column represents $RO = j$.

Serial Communication The finial thing to set up was how the computer was to receive data. This was done in python as decried in section subsection 3.2. The only difference was at the end of the python script was a statement to call a Matlab script to begin analysing the data.

References

[1] `www.mosis.com`

[2] A. Ferraiuolo, X. Zhang, and M. Tehranipoor, "Experimental Analysis of a Ring Oscillator Network for Hardware Trojan Detection in a 90nm ASIC," Int. Conf. on Computer-Aided Design (ICCAD), 2012

[3] `http://www.xilinx.com/support/download/index.htm`

[4] `http://www.digilentinc.com/Products/Detail.cfm?ProdADEPT2=`

[5] `http://www.digilentinc.com/classroom/Tutorials/`

[6] http://www.digilentinc.com/Data/Documents/Product\%20Documentation/VmodBB.pdf

[7] http://www.xilinx.com/support/documentation/ip_documentation/dcm_module.pdf

[8] http://www.xilinx.com/support/documentation/application_notes/xapp854.pdf

[9] http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS=

[10] http://www.digilentinc.com/Data/Products/ATLYS/Atlys_ISE_GPIO_UART.zip

[11] http://arduino.cc/playground/Interfacing/LinuxTTY